**Homework 9: Elastic MapReduce (Total Points: 100)**

Due: Monday November 12 11:59PM ET

**Problem 1 - Setup and Hive (50 points)**

Elastic MapReduce provides a very simple way of setting up a cluster of machines for the purpose of running big data analyses against an input source.

Step 1: Create an S3 bucket for your data and program

Create an S3 bucket with a name that makes it easily identifiable for this homework. Inside of the S3 bucket, add two folders: ‘data’, and ‘output’.

Step 2: Upload your job, and your data

Add the file ‘hive.q’ to your bucket. This is the script that will be run by EMR to perform some job. Download the following file, unpack it, and then upload it to your /data folder:

<http://ita.ee.lbl.gov/traces/NASA_access_log_Aug95.gz>

Step 3: Understand your data and your script

Before we set up a cluster to run a job, it’s important to understand what it is we are working with.

Take a look into the data file and you’ll see lines of data in the following format:

uplherc.upl.com - - [01/Aug/1995:00:00:08 -0400] "GET /images/ksclogo-medium.gif HTTP/1.0" 304 0

The data fields include:

* the hostname making the request
* time
* HTTP method
* requested path
* HTTP response code
* number of bytes in the reply

Next you want to take a look at the script that will be executed. This is a Hive script which allows us to write SQL type scripts to do MapReduce operations. The actual MapReduce implementations are transparent to us when using Hive, which makes it a (relatively) simple approach to running queries on large data sets.

**Step 4: Explain the Hive script (10 points)**

Provide comments to our hive.q script explaining what each section is doing. You do not have to go into deep detail here, just explain at a high level what actions are being performed.

|  |
| --- |
| -- We are making a new table (if one doesn’t exist), represented in json structure, with structure:  -- {source,date,request,status} for each row  CREATE EXTERNAL TABLE IF NOT EXISTS nasa\_logs ( Source STRING, datefield STRING, Request STRING, Status STRING )  -- We want to deserialize data into a table, using regular expressions.  -- We match the format of each line in access\_log\_Aug95 using regular expressions  -- and output into 4 columns ROW FORMAT SERDE "org.apache.hadoop.hive.serde2.RegexSerDe" WITH SERDEPROPERTIES( "input.regex" = "^([^ ]\*) - - \\[([^ ]\*[\\d]{4})[^ ]\* [^ ]\* \"([^ ]\* [^ ]\*) [^ ]\* ([\\d]{3}) [^ ]\*$", "output.format.string" = "%1$s %2$s %3$s %4$s" )  -- Specify the location of data from which to build the table.  -- This will be passed by EMR, and will be all contents of the data folder  -- of the s3 bucket, being the single file access\_log\_Aug95 LOCATION '${INPUT}';  -- Overwrite into the output directly from EMR (/output in the s3 bucket)  -- Only output the counts when the status column 4 = 404.  -- Will group by statuses, counting each row occurence of the status  -- Output will be : 404: Count of 404 occurences  INSERT OVERWRITE DIRECTORY '${OUTPUT}' select status, count(\*) AS total  from nasa\_logs where status = "404" group by status; |

**Step 5: Modify the Hive script (10 points)**

Without getting too complex, we’ll make fairly simple change to the above script in order to accomplish the following:

Display the count for each status type in the input file. Your results will show the counts for status types 200, 302, 304, etc.

Summarize your edits to the script here (you only need to provide the edited portion):

|  |
| --- |
| We will remove the filter ‘where status = “404”’ so that every status will be counted.  INSERT OVERWRITE DIRECTORY '${OUTPUT}' select status, count(\*) AS total  from nasa\_logs group by status; |

**Step 6: Run the Hive script (30 Points)**

Create an EMR cluster that includes Hadoop and Hive in its configuration. Add a step to run to your Hive script using the dataset uploaded to your S3 bucket.

Note: You can select to use the lowest cost machines available, which are c1.medium. It is recommended that you terminate on completion to minimize costs. Be aware that this cluster will probably take at least 20 minutes to run, and possibly over an hour.

Sometimes the formatting of the output can be a bit strange, you don’t need to worry about this.

Put the results of your job here when complete. We are interested in seeing the counts per status, the formatting is not a concern (so feel free to modify it by hand if you’d like to improve its readability):

|  |
| --- |
| **Status#Count**  200#1395806 302#26475 304#134138 400#2 403#171 404#9973 500#3 501#27 |

**Bonus 1: (5 Points)**

Using the data file <http://indeedeng.github.io/imhotep/files/worldcupplayerinfo_20140701.tsv> count the number of players per each age listed in the file.

Script source:

|  |
| --- |
|  |

**Problem 2: MapReduce (50 Points)**

For this problem we will use AMS EMR to build a cluster that we can access Hadoop through, and run MapReduce jobs. We will run a common wordcount on a large text file.

When you generate an EMR cluster, you configure what you wish to have provisioned on your instances (Hadoop, Hive, etc), and you essentially identify how much power you require in terms of how many nodes in your cluster and what machine types you wish to run. When you select to have machines built with Hadoop you get several sample files included by default. We will use one of those example files to run a wordcount.

Step 1:

Download the data file we will use for this exercise, the complete works of William Shakespeare in a single txt file:

<http://www.gutenberg.org/files/100/100-0.txt>

Step 2: Generate an EMR Cluster

Create an EMR cluster that you can SSH into directly. Make sure you identify a working key pair so that you can access your master node.

*Note:* EMR is pretty bad about telling you which instance types are going to work. If you get a cluster setup error, it may be because you selected a machine that is not supported. Also, in terms of machine size you may have trouble running medium sized instances. If you see errors about resources running out when you try to run a job, you might want to try increasing the size of your machine. You will probably need large machines for your cluster to work - make sure to kill all of them as soon as you are done.

Step 3: SSH into your master

Connect to your master node, and make sure to do so as the user ‘hadoop’. Keep this in mind - if you try to work with hadoop as user ‘ec2-user’ you will have trouble.

Once connected to your master node, traverse to /usr/lib/hadoop-mapreduce and take a look around. You will see that there are many jar files located here, including some samples.

We will be working with hadoop-mapreduce-examples.jar wordcount. The source code can be found online, but for now we are interested in wordcount:

<http://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>

**Step 4: Explain the code (20 Points)**

Setting up a working development environment is a fairly involved process, and can be frustrating when you encounter issues. We won’t try to do that here, but you should understand the basic operations of a MapReduce program. Add comments to the below code explaining what is going on:

|  |
| --- |
| import java.io.IOException; import java.util.StringTokenizer;  import org.apache.hadoop.conf.Configuration; import org.apache.hadoop.fs.Path; import org.apache.hadoop.io.IntWritable; import org.apache.hadoop.io.Text; import org.apache.hadoop.mapreduce.Job; import org.apache.hadoop.mapreduce.Mapper; import org.apache.hadoop.mapreduce.Reducer; import org.apache.hadoop.mapreduce.lib.input.FileInputFormat; import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class WordCount {  // The map function to run on each slave instance  // iterate on each individual word and return it in lines of  // [word] 1 for every occurence of each word  public static class TokenizerMapper  extends Mapper<Object, Text, Text, IntWritable>{   private final static IntWritable one = new IntWritable(1);  private Text word = new Text();   public void map(Object key, Text value, Context context  ) throws IOException, InterruptedException {  StringTokenizer itr = new StringTokenizer(value.toString());  while (itr.hasMoreTokens()) {  word.set(itr.nextToken());  context.write(word, one);  }  }  }  // The reducer function to run on the master instance when finished map on each slave node  // The reduce code is meant to perform some action on each key/value pair matching the passed in key, in this case, sum up each occurence.  // Output will be key, # of occurences for each key. The words from the map function are the keys.  public static class IntSumReducer  extends Reducer<Text,IntWritable,Text,IntWritable> {  private IntWritable result = new IntWritable();   public void reduce(Text key, Iterable<IntWritable> values,  Context context  ) throws IOException, InterruptedException {  int sum = 0;  for (IntWritable val : values) {  sum += val.get();  }  result.set(sum);  context.write(key, result);  }  }  // Main method of the java class, sets the map/reduce code, takes in the input/output paths from the CLI, sets the cofniguration for the run  public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  Job job = Job.getInstance(conf, "word count");  job.setJarByClass(WordCount.class);  job.setMapperClass(TokenizerMapper.class);  job.setCombinerClass(IntSumReducer.class);  job.setReducerClass(IntSumReducer.class);  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(IntWritable.class);  FileInputFormat.addInputPath(job, new Path(args[0]));  FileOutputFormat.setOutputPath(job, new Path(args[1]));  System.exit(job.waitForCompletion(true) ? 0 : 1);  } } |

Step 5: Count all words in the complete works of William Shakespeare

The code is written for you, the cluster is generated for you, and all you really need to do is set up and run the job. The steps for doing this are:

* put your input text (complete works of William Shakespeare) on your master node
* create an input directory *using hdfs*
* copy your input text into your input directory
* run the wordcount program against your input directory. The command will be something like: hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar <in> <out>

**Step 6: Demonstrate results (30 Points)**

Show that the job ran successfully and that you have results in your output directory. You can list your output results using something like: hadoop fs -ls <out>

Share the results here: **(15 points)**

|  |
| --- |
|  |

Share the first 20 lines of results from your first output file. The command will be something like:

hadoop fs -cat <out>/part-r-00000 | head -20

**(15 points)**

|  |
| --- |
|  |

**Bonus 2: (5 points)**

Modify the MapReduce program (or write your own in the language of your choice) to count how many times a word appears only once, how many times a word appears only twice, how many times a word only appears three times, and so on. The code must work.

Source:

|  |
| --- |
|  |